# Vocabulary Types for Composite Class Design

ISO/IEC JTC1 SC22 WG21 Programming Language C++

P3019R2

Working Group: Library Evolution, Library

Date: 2023-11-08

*Jonathan Coe <jonathanbcoe@gmail.com>*

*Antony Peacock <ant.peacock@gmail.com>*

*Sean Parent <sparent@adobe.com>*

## Abstract

We propose the addition of two new class templates to the C++ Standard Library: `indirect<T>` and `polymorphic<T>`.

These class templates have value semantics and compose well with other standard library types (such as vector) allowing the compiler to correctly generate special member functions.

The class template `indirect` confers value-like semantics on a free-store-allocated object. An `indirect` may hold an object of a class `T`. Copying the `indirect` will copy the object `T`. When a parent object contains a member of type `indirect<T>` and is accessed through a const access path, `const`ness will propagate from the parent object to the instance of `T` owned by the `indirect` member.

The class template `polymorphic` confers value-like semantics on a free-store-allocated object. A `polymorphic<T>` may hold an object of a class publicly derived from `T`. Copying the `polymorphic<T>` will copy the object of the derived type. When a parent object contains a member of type `polymorphic<T>` and is accessed through a const access path, `const`ness will propagate from the parent object to the instance of `T` owned by the `polymorphic` member.

This proposal is a fusion of two earlier individual proposals, P1950 and P0201. The design of the two proposed class templates is sufficiently similar that they should not be considered in isolation.

## Motivation

The standard library has no vocabulary type for a free-store-allocated object with value semantics. When designing a composite class, we may need an object to be stored indirectly to support incomplete types, reduce object size or support open-set polymorphism.

We propose the addition of two new class templates to the standard library to represent indirectly stored values: `indirect` and `polymorphic`. Both class templates represent free-store-allocated objects with value-like semantics. `polymorphic<T>` can own any object of a type publicly derived from `T`, allowing composite classes to contain polymorphic components. We require the addition of two classes to avoid the cost of virtual dispatch (calling the copy constructor of a potentially derived-type object through type erasure) when copying of polymorphic objects is not needed.

## Design requirements

We review the fundamental design requirements of `indirect` and `polymorphic` that make them suitable for composite class design.

### Special member functions

Both class templates are suitable for use as members of composite classes where the compiler will generate special member functions. This means that the class templates should provide the special member functions where they are supported by the owned object type `T`.

- `indirect<T, Alloc>` and `polymorphic<T, Alloc>` are default constructible in cases where `T` is default constructible.
- `indirect<T, Alloc>` is copy constructible where `T` is copy constructible and assignable.
- `polymorphic<T, Alloc>` is unconditionally copy constructible and assignable.
- `indirect<T, Alloc>` and `polymorphic<T, Alloc>` are unconditionally move constructible and assignable.
- `indirect<T, Alloc>` and `polymorphic<T, Alloc>` destroy the owned object in their destructors.

### Deep copies

Copies of `indirect<T>` and `polymorphic<T>` should own copies of the owned object created with the copy constructor of the owned object. In the case of `polymorphic<T>`, this means that the copy should own a copy of a potentially derived type object created with the copy constructor of the derived type object.

Note: Including a `polymorphic` component in a composite class means that virtual dispatch will be used (through type erasure) in copying the `polymorphic` member. Where a composite class contains a polymorphic member from a known set of types, prefer `std::variant` or `indirect<std::variant>` if indirect storage is required.

### `const` propagation

When composite objects contain `pointer`, `unique_ptr` or `shared_ptr` members they allow non-const access to their respective pointees when accessed through a const access path. This prevents the compiler from eliminating a source of const-correctness bugs and makes it difficult to reason about the const-correctness of a composite object.

Accessors of unique and shared pointers do not have const and non-const overloads:

```
T* unique_ptr<T>::operator->() const;
T& unique_ptr<T>::operator*() const;

T* shared_ptr<T>::operator->() const;
T& shared_ptr<T>::operator*() const;
```

When a parent object contains a member of type `indirect<T>` or `polymorphic<T>`, access to the owned object (of type `T`) through a const access path should be `const` qualified.

```
struct A {
    enum class Constness { CONST, NON_CONST };
    Constness foo() { return Constness::NON_CONST; }
    Constness foo() const { return Constness::CONST; };
};

class Composite {
    indirect<A> a_;

    Constness foo() { return a_.foo(); }
    Constness foo() const { return a_.foo(); };
};

int main() {
    Composite c;
    assert(c.foo() == A::Constness::NON_CONST);
    const Composite& cc = c;
    assert(cc.foo() == A::Constness::CONST);
}
```

### Value semantics

Both `indirect` and `polymorphic` are value types whose owned object is free-store-allocated (or some other memory resource controlled by the specified allocator).

When a value type is copied it gives rise to two independent objects that can be modified separately.

The owned object is part of the logical state of `indirect` and `polymorphic`. Operations on a const-qualified object do not make changes to the object's logical state nor to the logical state of other object.

`indirect<T>` and `polymorphic<T>` are default constructible in cases where `T` is default constructible. Moving a value type onto the free store should not add or remove the ability to be default constructed.

### Unobservable null state and interaction with `std::optional`

Both `indirect` and `polymorphic` have a null state that is used to implement move. The null state is not intended to be observable to the user. There is no `operator bool` or `has_value` member function. Accessing the value of an `indirect` or `polymorphic` after it has been moved from is erroneous behaviour. We provide a `valueless_after_move` member function that returns `true` if an object is in a valueless state. This allows explicit checks for the valueless state in cases where it cannot be verified statically.

Without a null state, moving `indirect` or `polymorphic` would require allocation and moving from the owned object. This would be expensive and would require the owned object to be moveable. The existence of a null state allows move to be implemented cheaply without requiring the owned object to be moveable.

Where a nullable `indirect` or `polymorphic` is required, using `std::optional` is recommended. This may become common practice since `indirect` and `polymorphic` can replace smart pointers in composite classes, where they are currently used to (mis)represent component objects. Putting `T` onto the free store should not make it nullable. Nullability must be explicitly opted into by using `std::optional<indirect<T>>` or `std::optional<polymorphic<T>>`.

`std::optional<>` is specialized for `indirect<>` and `polymorphic<>` so they incur no additional overhead.

Access to a `std::optional<indirect<T>>` or `std::optional<polymorphic<T>>` can be done with double indirection, `(**v)`, or with a single arrow operator to access a member, `v->some_member`.

Note: As the null state of `indirect` and `polymorphic` is not observable, and access to a moved-from object is erroneous, `std::optional` can be specialized by implementers to exchange pointers on move construction and assignment.

### Allocator support

Both `indirect` and `polymorphic` are allocator-aware types. They must be suitable for use in allocator-aware composite types and containers. Existing allocator-aware types in the standard, such as `vector` and `map`, take an allocator type as a template parameter, provide `allocator_type`, and have constructor overloads taking an additional `allocator_type_t` and allocator instance as arguments. As `indirect` and `polymorphic` need to work with and in the same way as existing allocator-aware types, they too take an allocator type as a template parameter, provide `allocator_type`, and have constructor overloads taking an additional `allocator_type_t` and allocator instance as arguments.

### Modelled types

The class templates `indirect` and `polymorphic` have strong similarities to existing class templates. These similarities motivate much of the design; we aim for consistency with existing library types, not innovation.

**Modelled types for `indirect`** The class template `indirect` owns an object of known type, permits copies, propagates const and is allocator aware.

- Like `optional` and `unique_ptr`, `indirect` can be in a valueless state; `indirect` can only get into the valueless state after move.
- `unique_ptr` and `optional` have preconditions for `operator->` and `operator*`: the behavior is undefined if `*this` does not contain a value.
- `unique_ptr` and `optional` mark `operator->` and `operator*` as noexcept: `indirect` does the same.
- `optional` and `indirect` know the underlying type of the owned object so can implement r-value qualified versions of `operator*`. For `unique_ptr` the underlying type is not known (it could be an instance of a derived class) so r-value qualified versions of `operator*` are not provided.
- Like `vector`, `indirect` owns an object created by an allocator. The move constructor and move assignment operator for `vector` are conditionally noexcept on properties of the allocator. Thus for `indirect`, the move constructor and move assignment operator for `indirect` are conditionally noexcept on properties of the allocator (Allocator instances may have different underlying memory resources, it is not possible for an allocator with one memory resource to delete an object in another memory resource. When allocators have different underlying memory resources, move necessitates the allocation of memory and cannot be marked noexcept). Like `vector`, `indirect` marks member and non-member swap as noexcept and requires allocators to be equal.
- Like `optional`, `indirect` knows the type of the owned object so forwards comparison operators and hash to the underlying object.
- Unlike `optional`, `indirect` is not observably valueless: use after move is erroneous. Formatting is supported by `indirect` by forwarding to the owned object.

**Modelled types for `polymorphic`** The class template `polymorphic` owns an object of known type, requires copies, propagates const and is allocator aware.

- Like `optional` and `unique_ptr`, `polymorphic` can be in a valueless state; `polymorphic` can only get into the valueless state after move.
- `unique_ptr` and `optional` have preconditions for `operator->` and `operator*`: the behavior is undefined if `*this` does not contain a value.
- `unique_ptr` and `optional` mark `operator->` and `operator*` as noexcept: `polymorphic` does the same.
- Neither `unique_ptr` nor `polymorphic` know the underlying type of the owned object so cannot implement r-value qualified versions of `operator*`. For `optional` the underlying type is known so r-value qualified versions of `operator*` are provided.
- Like `vector`, `polymorphic` owns an object created by an allocator. The move constructor and move assignment operator for `vector` are conditionally noexcept on properties of the allocator. Thus for `polymorphic`, the move constructor and move assignment operator for `polymorphic` are conditionally noexcept on properties of the allocator. Like `vector`, `polymorphic` marks member and non-member swap as noexcept and requires allocators to be equal.
- Like `unique_ptr`, `polymorphic` does not know the type of the owned object (it could be an instance of a derived type). As a result `polymorphic` cannot forward comparison operators, hash or formatting to the owned object.

### `noexcept` and narrow contracts

C++ library design guidelines recommend that member functions with narrow contracts (runtime-preconditions) should not be marked `noexcept`. This is partially motivated by a non-vendor implementation of the C++ standard library that uses exceptions in a debug build to check for precondition violations by throwing an exception. The `noexcept` status of `operator->` and `operator*` for `indirect` and `polymorphic` is identical to that of `optional` and `unique_ptr`. All have preconditions (`this` cannot be valueless), all are marked `noexcept`. Whatever strategy was used for testing `optional` and `unique_ptr` can be used for `indirect` and `polymorphic`.

Not marking `operator->` and `operator*` as `noexcept` for `indirect` and `polymorphic` would make them strictly less useful than `unique_ptr` in contexts where they would otherwise be a valid replacement.

### Design for polymorphic types

A type `PolymorphicInterface` used as a base class with `polymorphic` does not need a virtual destructor. The same mechanism that is used to call the copy constructor of a potentially derived-type object will be used to call the destructor.

To allow compiler-generation of special member functions of an abstract interface type `PolymorphicInterface` in conjunction with `polymorphic`, `PolymorphicInterface` needs at least a non-virtual protected destructor and a protected copy constructor. `PolymorphicInterface` does not need to be assignable, move constructible or move assignable for `polymorphic<PolymorphicInterface>` to be assignable, move constructible or move assignable.

```
class PolymorphicInterface {
  protected:
    PolymorphicInterface(const PolymorphicInterface&) = default;
    ~PolymorphicInterface() = default;
  public:
    // virtual functions
};
```

For an interface type with a public virtual destructor, users would potentially pay the cost of virtual dispatch twice when deleting `polymorphic<I>` objects containing derived-type objects.

All derived types owned by a `polymorphic` must be publicly copy constructible.

## Prior work

This proposal continues the work started in [P0201] and [P1950].

Previous work on a cloned pointer type [N3339] met with opposition because of the mixing of value and pointer semantics. We believe that the unambiguous value semantics of `indirect` and `polymorphic` as described in this proposal address these concerns.

## Impact on the standard

This proposal is a pure library extension. It requires additions to be made to the standard library header `<memory>`.

## Technical specifications

### X.Y Class template indirect [indirect]

**X.Y.1 Class template indirect general [indirect.general]** An *indirect value* is an object that manages the lifetime of an owned object. An indirect value object is *valueless* if it has no owned object. An indirect value may only become valueless after it has been moved from.

In every specialization `indirect<T, Allocator>`, the type `allocator_traits<Allocator>::value_type` shall be the same type as `T`. Every object of type `indirect<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed. The owned object shall be constructed using the function `allocator_traits<allocator_type>::rebind_traits<U>::construct` and destroyed using the function `allocator_traits<allocator_type>::rebind_traits<U>::destroy`, where `U` is either `allocator_type::value_type` or an internal type used by the indirect value.

Copy constructors for an indirect value obtain an allocator by calling `allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator belonging to the indirect value being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type& argument`. [Note 3:If an invocation of a constructor uses the default value of an optional allocator argument, then the allocator type must support value-initialization. end note] A copy of this allocator is used for any memory allocation and element construction performed by these constructors and by all member functions during the lifetime of each indirect value object, or until the allocator is replaced. The allocator may be replaced only via assignment or swap(). Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if (64.1) `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, (64.2) `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or (64.3) `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true within the implementation of the corresponding indirect value operation.

The template parameter `T` of `indirect` must be a non-union class type.

The template parameter `T` of `indirect` may be an incomplete type.

### X.Y.2 Class template indirect synopsis [indirect.syn]

```
template <class T, class Allocator = std::allocator<T>>
class indirect {
  T* p_; // exposition only
  Allocator allocator_; // exposition only
 public:
  using value_type = T;
  using allocator_type = Allocator;
  using pointer        = typename allocator_traits<Allocator>::pointer;
  using const_pointer  = typename allocator_traits<Allocator>::const_pointer;

  constexpr indirect();

  template <class... Ts>
  explicit constexpr indirect(Ts&&... ts);

  template <class... Ts>
  constexpr indirect(
```

```
  std::allocator_arg_t, const Allocator& alloc, Ts&&... ts);

constexpr indirect(const indirect& other);

constexpr indirect(std::allocator_arg_t, const Allocator& alloc,
                   const indirect& other);

constexpr indirect(indirect&& other) noexcept(see below);

constexpr indirect(std::allocator_arg_t, const Allocator& alloc,
                   indirect&& other) noexcept(see below);

constexpr ~indirect();

constexpr indirect& operator=(const indirect& other);

constexpr indirect& operator=(indirect&& other) noexcept(see below);

constexpr const T& operator*() const & noexcept;

constexpr T& operator*() & noexcept;

constexpr const T&& operator*() const && noexcept;

constexpr T&& operator*() && noexcept;

constexpr const_pointer operator->() const noexcept;

constexpr pointer operator->() noexcept;

constexpr bool valueless_after_move() const noexcept;

constexpr allocator_type get_allocator() const noexcept;

constexpr void swap(indirect& other) noexcept;

friend constexpr void swap(indirect& lhs, indirect& rhs) noexcept;

template <class U, class AA>
friend constexpr auto operator==(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
friend constexpr auto operator!=(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);
```

```
template <class U, class AA>
friend constexpr auto operator<(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
friend constexpr auto operator<=(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
friend constexpr auto operator>(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
friend constexpr auto operator>=(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
friend constexpr auto operator<=>(
  const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U>
friend constexpr auto operator==(const indirect<T, A>& lhs, const U& rhs);

template <class U>
friend constexpr auto operator==(const U& lhs, const indirect<T, A>& rhs);

template <class U>
friend constexpr auto operator!=(const indirect<T, A>& lhs, const U& rhs);

template <class U>
friend constexpr auto operator!=(const U& lhs, const indirect<T, A>& rhs);

template <class U>
friend constexpr auto operator<(const indirect<T, A>& lhs, const U& rhs);

template <class U>
friend constexpr auto operator<(const U& lhs, const indirect<T, A>& rhs);

template <class U>
friend constexpr auto operator<=(const indirect<T, A>& lhs, const U& rhs);

template <class U>
friend constexpr auto operator<=(const U& lhs, const indirect<T, A>& rhs);

template <class U>
friend constexpr auto operator>(const indirect<T, A>& lhs, const U& rhs);
```

```cpp
  template <class U>
  friend constexpr auto operator>(const U& lhs, const indirect<T, A>& rhs);

  template <class U>
  friend constexpr auto operator>=(const indirect<T, A>& lhs, const U& rhs);

  template <class U>
  friend constexpr auto operator>=(const U& lhs, const indirect<T, A>& rhs);

  template <class U>
  friend constexpr auto operator<=>(const indirect<T, A>& lhs, const U& rhs);

  template <class U>
  friend constexpr auto operator<=>(const U& lhs, const indirect<T, A>& rhs);
};

template <class T, class Alloc>
struct hash<indirect<T, Alloc>>;
```

### X.Y.3 Constructors [indirect.ctor]

```cpp
constexpr indirect()
```

- *Mandates*: `is_default_constructible_v<T>` is true.
- *Effects*: Constructs an indirect owning a default-constructed `T`. `allocator_` is default constructed.
- *Postconditions*: `*this` is not valueless.

```cpp
template <class... Ts>
explicit constexpr indirect(Ts&&... ts);
```

- *Constraints*: `is_constructible_v<T, Ts...>` is true.
- *Effects*: Constructs an indirect owning an instance of `T` created with the arguments `Ts`. `allocator_` is default constructed.
- *Postconditions*: `*this` is not valueless.

```cpp
template <class... Ts>
constexpr indirect(std::allocator_arg_t, const Allocator& alloc, Ts&&... ts);
```

- *Constraints*: `is_constructible_v<T, Ts...>` is true.
- *Preconditions*: `Allocator` meets the *Cpp17Allocator* requirements.
- *Effects*: Equivalent to the preceding constructor except that the allocator is initialized with alloc. `allocator_` is initialized with `alloc`.
- *Postconditions*: `*this` is not valueless.

```
constexpr indirect(const indirect& other);
```

- *Mandates*: `is_copy_constructible_v<T>` is true.
- *Preconditions*: `other` is not valueless.
- *Effects*: Constructs an indirect owning an instance of `T` created with the copy constructor of the object owned by `other`. `allocator` is obtained by calling `allocator_traits<allocator_type>::select_on_container_copy_construction`on the allocator belonging to the object being copied.
- *Postconditions*: `*this` is not valueless.

```
constexpr indirect(std::allocator_arg_t, const Allocator& alloc,
                   const indirect& other);
```

- *Mandates*: `is_copy_constructible_v<T>` is true.
- *Preconditions*: `other` is not valueless and `Allocator` meets the *Cpp17Allocator* requirements.
- *Effects*: Equivalent to the preceding constructor except that the allocator is initialized with alloc.
- *Postconditions*: `*this` is not valueless.

```
constexpr indirect(indirect&& other) noexcept;
```

- *Preconditions*: `other` is not valueless.
- *Effects*: Constructs an `indirect` owning the object owned by `other`. `allocator` is created by move construction from the allocator belonging to the object being moved.
- *Postconditions*: `other` is valueless.
- *Remarks*: This constructor does not require that `is_move_constructible_v<T>` is true.

```
constexpr indirect(std::allocator_arg_t, const Allocator& alloc,
                   indirect&& other) noexcept;
```

- *Preconditions*: `other` is not valueless and `Allocator` meets the *Cpp17Allocator* requirements.
- *Effects*: Equivalent to the preceding constructors except that the allocator is initialized with alloc.
- *Postconditions*: `other` is valueless.
- *Remarks*: This constructor does not require that `is_move_constructible_v<T>` is true.

### X.Y.4 Destructor [indirect.dtor]

```
constexpr ~indirect();
```

- *Effects*: If `*this` is not valueless, destroys the owned object.

### X.Y.5 Assignment [indirect.assign]

```
constexpr indirect& operator=(const indirect& other);
```

- *Mandates*: `is_copy_assignable_v<T>` and `is_copy_constructible_v<T>`is true.

- *Preconditions*: `other` is not valueless.

- *Effects*: If `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::valu` `== true`, `allocator` is set to the allocator of `other`. If allocator is not changed, `std::is_copy_assignable_v<T>` is true, and `*this` is not valueless, copy assigns the owned object in `*this` from the owned object in `other`. Otherwise, destroys the owned object, if any, then copy constructs a new object using the object owned by `other`.

- *Postconditions*: `*this` is not valueless.

```
constexpr indirect& operator=(indirect&& other) noexcept(
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
```

*Mandates*: `is_move_constructible_v<T>`is true.

- *Preconditions*: `other` is not valueless.

- *Effects*: If `allocator_traits<allocator_type>::propagate_on_container_move_assignment::valu` `== true`, `allocator` is set to the allocator of `other`. If allocator is propagated or is equal to the allocator of `other`, destroys the owned object, if any, then takes ownership of the object owned by `other`. Otherwise, destroys the owned object, if any, then move constructs an object from the object owned by `other`.

- *Postconditions*: `*this` is not valueless. `other` is valueless.

### X.Y.6 Observers [indirect.observers]

```
constexpr const T& operator*() const & noexcept;
constexpr T& operator*() & noexcept;
constexpr const T&& operator*() const && noexcept;
constexpr T&& operator*() && noexcept;
```

- *Preconditions*: `*this` is not valueless.

- *Effects*: Returns a reference to the owned object.

- *Remarks*: These functions are constexpr functions.

```
constexpr const_pointer operator->() const noexcept;
constexpr pointer operator->() noexcept;
```

- *Preconditions*: `*this` is not valueless.
- *Effects*: Returns a pointer to the owned object.
- *Remarks*: These functions are constexpr functions.

```
constexpr bool valueless_after_move() const noexcept;
```

- *Returns*: `true` if `*this` is valueless, otherwise `false`.

```
constexpr allocator_type get_allocator() const noexcept;
```

- *Returns*: A copy of the Allocator object used to construct the owned object.

### X.Y.7 Swap [indirect.swap]

```
constexpr void swap(indirect& other) noexcept(
  allocator_traits::propagate_on_container_swap::value
  || allocator_traits::is_always_equal::value);
```

- *Preconditions*: `*this` is not valueless, `other` is not valueless.
- *Effects*: Swaps the objects owned by `*this` and `other`. If `allocator_traits<allocator_type>::propaga` is `true`, then allocator_type shall meet the *Cpp17Swappable* requirements and the allocators of `*this` and `other` shall also be exchanged by calling `swap` as described in [swappable.requirements]. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `*this.get_allocator() == other.get_allocator()`.
- *Remarks*: Does not call `swap` on the owned objects directly.

```
constexpr void swap(indirect& lhs, indirect& rhs);
```

- *Effects*: Equivalent to `lhs.swap(rhs)`.

### X.Y.8 Relational operators [indirect.rel]

```
template <class U, class AA>
constexpr auto operator==(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
constexpr auto operator!=(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
constexpr auto operator<(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
constexpr auto operator<=(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);
```

```
template <class U, class AA>
constexpr auto operator>(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
constexpr auto operator>=(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);

template <class U, class AA>
constexpr auto operator<=>(const indirect<T, A>& lhs, const indirect<U, AA>& rhs);
```

- *Constraints*: The operator `op` is defined for `T`.
- *Preconditions*: `lhs` is not valueless, `rhs` is not valueless.
- *Effects*: Returns `*lhs` *op* `*rhs`.
- *Remarks*: Specializations of this function template for which `*lhs` *op* `*rhs` is a core constant expression are constexpr functions.

### X.Y.9 Comparison with T [indirect.comp.with.t]

```
template <class U>
constexpr auto operator==(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator!=(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator<(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator<=(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator>(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator>=(const indirect<T, A>& lhs, const U& rhs);

template <class U>
constexpr auto operator<=>(const indirect<T, A>& lhs, const U& rhs);
```

- *Constraints*: The operator `op` is defined for `T`.
- *Preconditions*: `lhs` is not valueless.
- *Effects*: Returns `*lhs` *op* `rhs`.
- *Remarks*: Specializations of this function template for which `*lhs` *op* `rhs` is a core constant expression, are constexpr functions.

```
template <class U>
constexpr auto operator==(const U& lhs, const indirect<T, A>& rhs);

template <class U>
constexpr auto operator!=(const U& lhs, const indirect<T, A>& rhs);
```

```
template <class U>
constexpr auto operator<(const U& lhs, const indirect<T, A>& rhs);

template <class U>
constexpr auto operator<=(const U& lhs, const indirect<T, A>& rhs);

template <class U>
constexpr auto operator>(const U& lhs, const indirect<T, A>& rhs);

template <class U>
constexpr auto operator>=const U& lhs, const indirect<T, A>& rhs);

template <class U>
constexpr auto operator<=>(const U& lhs, const indirect<T, A>& rhs);
```

- *Constraints*: The operator `op` is defined for `T`.
- *Preconditions*: `rhs` is not valueless.
- *Effects*: Returns `lhs` *op* `*rhs`.
- *Remarks*: Specializations of this function template for which `lhs` *op* `*rhs` is a core constant expression, are constexpr functions.

### X.Y.10 Hash support [indirect.hash]

```
template <class T, class Alloc>
struct std::hash<indirect<T, Alloc>>;
```

- *Preconditions*: `i` is not valueless.

The specialization `hash<indirect<T, Alloc>>` is enabled ([unord.hash]) if and only if `hash<remove_const_t<T>>` is enabled. When enabled for an object `i` of type `indirect<T, Alloc>`, then `hash<indirect<T, Alloc>>()(i)` evaluates to the same value as `hash<remove_const_t<T>>()(*i)`. The member functions are not guaranteed to be noexcept.

### X.Y.12 Optional support [indirect.optional]

```
template <class T, class Alloc>
class std::optional<indirect<T, Alloc>>;
```

The specialization `std::optional<indirect<T, Alloc>>` guarantees `sizeof(std::optional<indirect<T, Alloc>>) == sizeof(indirect<T, Alloc>>)`.

```
// [optional.observe], observers
constexpr const indirect<T, Alloc>& operator->() const noexcept;
constexpr indirect<T, Alloc>& operator->() noexcept;
```

- *Preconditions*: `*this` contains a value. The contained indirect value is not valueless.

- *Returns*: `val`.
- *Remarks*: These functions are constexpr. The specialization `std::optional<indirect<T, Alloc>>` provides `operator->` that returns a reference to the contained `indirect`.

Otherwise, the interface of the specialization is as defined in [optional].

#### X.Y.13 Formatter support [indirect.fmt]

```
// [indirect.fmt]
template <class T, class Alloc, class charT>
struct std::formatter<indirect<T, Alloc>, charT> : std::formatter<T, charT> {
  template<class ParseContext>
  constexpr typename ParseContext::iterator parse(ParseContext& ctx);

  template<class FormatContext>
  typename FormatContext::iterator format(
    indirect<T, Alloc> const& value, FormatContext& ctx) const;
};
```

Specialization of `std::formatter<indirect<T, Alloc>, charT>` when the underlying `T` supports specialisation of `std::formatter<T, charT>`.

- Preconditions: `value` is not valueless. The specialization `formatter<T, charT>` meets the *Formatter* requirements.

## Feature-test Macro [indirect.predefined.ft]

Add a new feature-test macro:

```
#define __cpp_lib_indirect 2023XXL
```

#### X.Z Class template polymorphic [polymorphic]

**X.Z.1 Class template polymorphic general [polymorphic.general]** A *polymorphic value* is an object that manages the lifetime of an owned object. A polymorphic value object may own objects of different types at different points in its lifetime. A polymorphic value object is *valueless* if it has no owned object. A polymorphic value may only become valueless after it has been moved from.

In every specialization `polymorphic<T, Allocator>`, the type `allocator_traits<Allocator>::value_type` shall be the same type as `T`. Every object of type `polymorphic<T, Allocator>` uses an object of type `Allocator` to allocate and free storage for the owned object as needed. The owned object shall be constructed using the function `allocator_traits<allocator_type>::rebind_traits<U>::construct` and destroyed using the function `allocator_traits<allocator_type>::rebind_traits<U>::destroy`, where `U` is either `allocator_type::value_type` or an internal type used by the polymorphic value.

Copy constructors for a polymorphic value obtain an allocator by calling `allocator_traits<allocator_type>::select_on_container_copy_construction` on the allocator belonging to the polymorphic value being copied. Move constructors obtain an allocator by move construction from the allocator belonging to the container being moved. Such move construction of the allocator shall not exit via an exception. All other constructors for these container types take a `const allocator_type& argument`. [Note 3:If an invocation of a constructor uses the default value of an optional allocator argument, then the allocator type must support value-initialization. end note] A copy of this allocator is used for any memory allocation and element construction performed by these constructors and by all member functions during the lifetime of each polymorphic value object, or until the allocator is replaced. The allocator may be replaced only via assignment or swap(). Allocator replacement is performed by copy assignment, move assignment, or swapping of the allocator only if (64.1) `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, (64.2) `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or (64.3) `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true within the implementation of the corresponding polymorphic value operation.

The template parameter `T` of `polymorphic` must be a non-union class type.

The template parameter `T` of `polymorphic` may be an incomplete type.

### X.Z.2 Class template polymorphic synopsis [polymorphic.syn]

```
template <class T, class Allocator = std::allocator<T>>
class polymorphic {
  control_block* control_block_; // exposition only
  Allocator allocator_; // exposition only
 public:
  using value_type = T;
  using allocator_type = Allocator;
  using pointer        = typename allocator_traits<Allocator>::pointer;
  using const_pointer  = typename allocator_traits<Allocator>::const_pointer;

  constexpr polymorphic();

  template <class U, class... Ts>
  explicit constexpr polymorphic(std::in_place_type_t<U>, Ts&&... ts);

  template <class U, class... Ts>
  constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                        std::in_place_type_t<U>, Ts&&... ts);

  constexpr polymorphic(const polymorphic& other);
```

```cpp
  constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                        const polymorphic& other);

  constexpr polymorphic(polymorphic&& other) noexcept(see below);

  constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                        polymorphic&& other) noexcept(see below);

  constexpr ~polymorphic();

  constexpr polymorphic& operator=(const polymorphic& other);

  constexpr polymorphic& operator=(polymorphic&& other) noexcept(see below);

  constexpr const T& operator*() const noexcept;

  constexpr T& operator*() noexcept;

  constexpr const_pointer operator->() const noexcept;

  constexpr pointer operator->() noexcept;

  constexpr bool valueless_after_move() const noexcept;

  constexpr allocator_type get_allocator() const noexcept;

  constexpr void swap(polymorphic& other) noexcept(see below);

  friend constexpr void swap(polymorphic& lhs, polymorphic& rhs) noexcept(see below);
};
```

### X.Z.3 Constructors [polymorphic.ctor]

```cpp
constexpr polymorphic()
```

- *Mandates*: `is_default_constructible_v<T>` is true, `is_copy_constructible_v<T>` is true.
- *Effects*: Constructs a polymorphic owning a default-constructed `T`. `allocator_` is default constructed.
- *Postconditions*: `*this` is not valueless.

```cpp
template <class U, class... Ts>
explicit constexpr polymorphic(std::in_place_type_t<U>, Ts&&... ts);
```

- *Constraints*: `is_base_of_v<T, U>` is true, `is_constructible_v<U, Ts...>` is true, `is_copy_constructible_v<U>` is true.

- *Effects*: Constructs a polymorphic owning an instance of `U` created with the arguments `Ts`. `allocator_` is default constructed.

- *Postconditions*: `*this` is not valueless.

```
template <class U, class... Ts>
constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                      std::in_place_type_t<U>, Ts&&... ts);
```

- *Constraints*: `is_base_of_v<T, U>` is true, `is_constructible_v<U, Ts...>` is true, `is_copy_constructible_v<U>` is true.

- *Preconditions*: `Allocator` meets the *Cpp17Allocator* requirements.

- *Effects*: Equivalent to the preceding constructor except that the `allocator_` is initialized with `alloc`.

- *Postconditions*: `*this` is not valueless.

```
constexpr polymorphic(const polymorphic& other);
```

- *Preconditions*: `other` is not valueless.

- *Effects*: Constructs a polymorphic owning an instance of `T` created with the copy constructor of the object owned by `other`. `allocator` is obtained by calling `allocator_traits<allocator_type>::select_on_container_copy_construction`on the allocator belonging to the object being copied.

- *Postconditions*: `*this` is not valueless.

```
constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                      const polymorphic& other);
```

- *Preconditions*: `other` is not valueless and `Allocator` meets the *Cpp17Allocator* requirements.

- *Effects*: Equivalent to the preceding constructor except that the allocator is initialized with alloc.

- *Postconditions*: `*this` is not valueless.

```
constexpr polymorphic(polymorphic&& other) noexcept;
```

- *Preconditions*: `other` is not valueless.

- *Effects*: Constructs a polymorphic that takes ownership of the object owned by `other`. `allocator` is created by move construction from the allocator belonging to the object being moved.

- *Postconditions*: `other` is valueless.

- *Remarks*: This constructor does not require that `is_move_constructible_v<T>` is true.

```
constexpr polymorphic(std::allocator_arg_t, const Allocator& alloc,
                      polymorphic&& other) noexcept;
```

- *Preconditions*: `other` is not valueless and `Allocator` meets the *Cpp17Allocator* requirements.
- *Effects*: Equivalent to the preceding constructor except that the allocator is initialized with alloc.
- *Postconditions*: `other` is valueless.
- *Remarks*: This constructor does not require that `is_move_constructible_v<T>` is true.

### X.Z.4 Destructor [polymorphic.dtor]

```
constexpr ~polymorphic();
```

- *Effects*: If `*this` is not valueless, destroys the owned object.

### X.Z.5 Assignment [polymorphic.assign]

```
constexpr polymorphic& operator=(const polymorphic& other);
```

- *Preconditions*: `other` is not valueless.
- *Effects*: If `*this` is not valueless, destroys the owned object. If `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value == true`, `allocator` is set to the allocator of `other`. Copy constructs a new object using the object owned by `other`.
- *Postconditions*: `*this` is not valueless.

```
constexpr polymorphic& operator=(polymorphic&& other) noexcept(
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);
```

- *Preconditions*: `other` is not valueless.
- *Effects*: If `allocator_traits<allocator_type>::propagate_on_container_move_assignment::valu == true`, `allocator` is set to the allocator of `other`. If allocator is propagated or is equal to the allocator of `other`, destroys the owned object, if any, then takes ownership of the object owned by `other`. Otherwise, destroys the owned object, if any, then move constructs an object from the object owned by `other`.
- *Postconditions*: `*this` is not valueless. `other` is valueless.

### X.Z.6 Observers [polymorphic.observers]

```
constexpr const T& operator*() const noexcept;
constexpr T& operator*() noexcept;
```

- *Preconditions*: `*this` is not valueless.
- *Effects*: Returns a reference to the owned object.

- *Remarks*: These functions are constexpr functions.

```
constexpr const_pointer operator->() const noexcept;
constexpr pointer operator->() noexcept;
```

- *Preconditions*: `*this` is not valueless.
- *Effects*: Returns a pointer to the owned object.
- *Remarks*: These functions are constexpr functions.

```
constexpr bool valueless_after_move() const noexcept;
```

- *Returns*: `true` if `*this` is valueless, otherwise `false`.

```
constexpr allocator_type get_allocator() const noexcept;
```

- *Returns*: A copy of the Allocator object used to construct the owned object.

### X.Z.7 Swap [polymorphic.swap]

```
constexpr void swap(polymorphic& other) noexcept(
  allocator_traits::propagate_on_container_swap::value
  || allocator_traits::is_always_equal::value);
```

- *Preconditions*: `*this` is not valueless, `other` is not valueless.
- *Effects*: Swaps the objects owned by `*this` and `other`. If `allocator_traits<allocator_type>::propaga` is `true`, then allocator_type shall meet the *Cpp17Swappable* requirements and the allocators of `*this` and `other` shall also be exchanged by calling `swap` as described in [swappable.requirements]. Otherwise, the allocators shall not be swapped, and the behavior is undefined unless `*this.get_allocator() == other.get_allocator()`.
- *Remarks*: Does not call `swap` on the owned objects directly.

```
constexpr void swap(polymorphic& lhs, polymorphic& rhs);
```

- *Effects*: Equivalent to `lhs.swap(rhs)`.

### X.Z.8 Optional support [polymorphic.optional]

```
template <class T, class Alloc>
class std::optional<polymorphic<T, Alloc>>;
```

The specialization `std::optional<polymorphic<T, Alloc>>` guarantees `sizeof(std::optional<polymorphic<T, Alloc>>) == sizeof(polymorphic<T, Alloc>>)`.

```
// [optional.observe], observers
constexpr const polymorphic<T, Alloc>& operator->() const noexcept;
constexpr polymorphic<T, Alloc>& operator->() noexcept;
```

- *Preconditions*: `*this` is not valueless. The contained polymorphic value is not valueless.
- *Returns*: `val`.
- *Remarks*: These functions are constexpr. The specialization `std::optional<polymorphic<T, Alloc>>` provides `operator->` that returns a reference to the contained `polymorphic`.

Otherwise, the interface of the specialization is as defined in [optional].

### Feature-test Macro [polymorphic.predefined.ft]

Add a new feature-test macro:

```
#define __cpp_lib_polymorphic 2023XXL
```

### Reference implementation

A C++20 reference implementation of this proposal is available on GitHub at [https://www.github.com/jbcoe/value_types].

### Acknowledgements

The authors would like to thank Andrew Bennieston, Josh Berne, Bengt Gustafsson, Casey Carter, Rostislav Khlebnikov, Daniel Krugler, David Krauss, Ed Catmur, Geoff Romer, German Diago, Jonathan Wakely, Kilian Henneberger, LanguageLawyer, Louis Dionne, Maciej Bogus, Malcolm Parsons, Matthew Calabrese, Nathan Myers, Neelofer Banglawala, Nevin Liber, Nina Ranns, Patrice Roy, Roger Orr, Stephan T Lavavej, Stephen Kelly, Thomas Koeppe, Thomas Russell, Tom Hudson, Tomasz Kaminski, Tony van Eerd and Ville Voutilainen for suggestions and useful discussion.

### References

"*A Preliminary Proposal for a Deep-Copying Smart Pointer*", W. E. Brown, 2012 [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3339.pdf]

*A polymorphic value-type for C++*, J. B. Coe, S. Parent 2019 [https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0201r6.html]

*A Free-Store-Allocated Value Type for C++*, J. B. Coe, A. Peacock 2022 [https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1950r2.html]

A C++20 reference implementation is available on GitHub [https://www.github.com/jbcoe/value_types]

### Appendix A: Detailed design decisions

We discuss some of the decisions that were made in the design of `indirect` and `polymorphic`. Where there are multiple options, we discuss the advantages and disadvantages of each.

### Two class templates, not one

It is conceivable that a single class template could be used as a vocabulary type for an indirect value type supporting polymorphism. However, implementing this would impose efficiency costs on the copy constructor when the owned object is the same type as the template type. When the owned object is a derived type, the copy constructor uses type erasure to perform dynamic dispatch and call the derived type copy constructor. The overhead of indirection and a virtual function call is not tolerable where the owned object type and template type match.

One potential solution would be to use a `std::variant` to store the owned type or the control block used to manage the owned type. This would allow the copy constructor to be implemented efficiently when the owned type and template type match. This would increase the object size beyond that of a single pointer as the discriminant must be stored.

For the sake of minimal size and efficiency, we opted to use two class templates.

### Copiers, deleters, pointer constructors, and allocator support

The older types `indirect_value` and `polymorphic_value` had constructors that take a pointer, copier, and deleter. The copier and deleter could be used to specify how the object should be copied and deleted. The existence of a pointer constructor introduces undesirable properties into the design of `polymorphic_value`, such as allowing the possibility of object slicing on copy when the dynamic and static types of a derived-type pointer do not match.

We decided to remove the copier, delete, and pointer constructor in favour of adding allocator support. A pointer constructor and support for custom copiers and deleters are not core to the design of either class template; both could be added in a later revision of the standard if required.

We have been advised that allocator support must be a part of the initial implementation and cannot be added retrospectively. As `indirect` and `polymorphic` are intended to be used alongside other C++ standard library types, such as `std::map` and `std::vector`, it is important that they have allocator support in contexts where allocators are used.

### Pointer-like helper functions

Earlier revisions of `polymorphic_value` had helper functions to get access to the underlying pointer. These were removed under the advice of the Library Evolution Working Group as they were not core to the design of the class template, nor were they consistent with value-type semantics.

Pointer-like accessors like `dynamic_pointer_cast` and `static_pointer_cast`, which are provided for `std::shared_ptr`, could be added in a later revision of the standard if required.

### Implicit conversions

We decided that there should be no implicit conversion of a value `T` to an `indirect<T>` or `polymorphic<T>`. An implicit conversion would require using the free store and memory allocation, which is best made explicit by the user.

```
Rectangle r(w, h);
polymorphic<Shape> s = r; // error
```

To transform a value into `indirect` or `polymorphic`, the user must use the appropriate constructor.

```
Rectangle r(w, h);
polymorphic<Shape> s(std::in_place_type<Rectangle>, r);
assert(dynamic_cast<Rectangle*>(&*s) != nullptr);
```

### Explicit conversions

The older class template `polymorphic_value` had explicit conversions, allowing construction of a `polymorphic_value<T>` from a `polymorphic_value<U>`, where `T` was a base class of `U`.

```
polymorphic_value<Quadrilateral> q(std::in_place_type<Rectangle>, w, h);
polymorphic_value<Shape> s = q;
assert(dynamic_cast<Rectangle*>(&*s) != nullptr);
```

Similar code cannot be written with `polymorphic` as it does not allow conversions between derived types:

```
polymorphic<Quadrilateral> q(std::in_place_type<Rectangle>, w, h);
polymorphic<Shape> s = q; // error
```

This is a deliberate design decision. `polymorphic` is intended to be used for ownership of member data in composite classes where compiler-generated special member functions will be used.

There is no motivating use case for explicit conversion between derived types outside of tests.

A converting constructor could be added in a future version of the C++ standard.

### Comparisons returning `auto`

We opt to return `auto` from comparsion operators on `indirect<T>` so that the return type perfectly matches that of the underlying comparison for `T`. While deferring the return type to the underlying type does support unusual user-defined comparsion operators, we prefer to do so rather than impose requirements on the user-defined operators for consistency. Adoption of indirect or moving an object onto the heap should not be impeded by unusual choices for the return type of comparison operators on user-defined types.

### Supporting `operator()` `operator[]`

There is no need for `indirect` or `polymorphic` to provide a function call or an indexing operator. Users who wish to do that can just access the value and call its operator. Furthermore, unlike comparisons, function calls or indexing operators do not compose further; for example, a composite would not be able to automatically generate a composited `operator()` or an `operator[]`.

### Member function `emplace`

Neither `indirect` nor `polymorphic` support `emplace` as a member function. The member function `emplace` could be added as :

```
template <typename ...Ts>
indirect::emplace(Ts&& ...ts);
```

```
template <typename U, typename ...Ts>
polymorphic::emplace(in_place_type<U>, Ts&& ...ts);
```

This would be API noise. It offers no efficiency improvement over:

```
some_indirect = indirect(/* arguments */);
```

```
some_polymorphic = polymorphic(in_place_type<U>, /* arguments */);
```

### Small Buffer Optimisation

It is possible to implement `polymorphic` with a small buffer optimisation, similar to that used in `std::function`. This would allow `polymorphic` to store small objects without allocating memory. Like `std::function`, the size of the small buffer is left to be specified by the implementation.

The authors are sceptical of the value of a small buffer optimisation for objects from a type hierarchy. If the buffer is too small, all instances of `polymorphic` will be larger than needed. This is because they will allocate heap in addition to having the memory from the (empty) buffer as part of the object size. If the buffer is too big, `polymorphic` objects will be larger than necessary, potentially introducing the need for `indirect<polymorphic<T>>`.

We could add a non-type template argument to `polymorphic` to specify the size of the small buffer:

```
template <typename T, typename Alloc, size_t BufferSize>
class polymorphic;
```

However, we opt not to do this to maintain consistency with other standard library types. Both `std::function` and `std::string` leave the buffer size as an implementation detail. Including an additional template argument in a later revision of the standard would be a breaking change. With usage experience, implementers will be able to determine if a small buffer optimisation is worthwhile, and what the optimal buffer size might be.

A small buffer optimisation makes little sense for `indirect` as the sensible size of the buffer would be dictated by the size of the stored object. This removes support for incomplete types and locates storage for the object locally, defeating the purpose of `indirect`.

## Appendix B: Before and after examples

We include some minimal, illustrative examples of how `indirect` and `polymorphic` can be used to simplify composite class design.

### Using `indirect` for binary compatibility using the PIMPL idiom

Without `indirect`, we use `std::unique_ptr` to manage the lifetime of the implementation object. All const-qualified methods of the composite will need to be manually checked to ensure that they are not calling non-const qualified methods of component objects.

#### Before, without using `indirect`

```
// Class.h

class Class {
  class Impl;
  std::unique_ptr<Impl> impl_;
 public:
  Class();
  ~Class();
  Class(const Class&);
  Class& operator=(const Class&);
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;

  void do_something();
};
```

```
// Class.cpp

class Impl {
 public:
  void do_something();
};
```

```
Class::Class() : impl_(std::make_unique<Impl>()) {}

Class::~Class() = default;

Class::Class(const Class& other) : impl_(std::make_unique<Impl>(*other.impl_)) {}

Class& Class::operator=(const Class& other) {
  if (this != &other) {
    Class tmp(other);
    using std::swap;
    swap(*this, tmp);
  }
  return *this;
}

Class(Class&&) noexcept = default;
Class& operator=(Class&&) noexcept = default;

void Class::do_something() {
  impl_->do_something();
}
```

**After, using indirect**

```
// Class.h

class Class {
  indirect<class Impl> impl_;
 public:
  Class();
  ~Class();
  Class(const Class&);
  Class& operator=(const Class&);
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;

  void do_something();
};
```

```
// Class.cpp

class Impl {
 public:
  void do_something();
};
```

```
Class::Class() : impl_(indirect<Impl>()) {}
Class::~Class() = default;
Class::Class(const Class&) = default;
Class& Class::operator=(const Class&) = default;
Class(Class&&) noexcept = default;
Class& operator=(Class&&) noexcept = default;

void Class::do_something() {
  impl_->do_something();
}
```

### Using `polymorphic` for a composite class

Without `polymorphic`, we use `std::unique_ptr` to manage the lifetime of component objects. All const-qualified methods of the composite will need to be manually checked to ensure that they are not calling non-const qualified methods of component objects.

### Before, without using `polymorphic`

```
class Canvas;

class Shape {
 public:
  virtual ~Shape() = default;
  virtual std::unique_ptr<Shape> clone() = 0;
  virtual void draw(Canvas&) const = 0;
};

class Picture {
  std::vector<std::unique_ptr<Shape>> shapes_;

 public:
  Picture(const std::vector<std::unique_ptr<Shape>>& shapes) {
    shapes_.reserve(shapes.size());
    for (auto& shape : shapes) {
      shapes_.push_back(shape->clone());
    }
  }

  Picture(const Picture& other) {
    shapes_.reserve(other.shapes_.size());
    for (auto& shape : other.shapes_) {
      shapes_.push_back(shape->clone());
    }
  }
```

```
  Picture& operator=(const Picture& other) {
    if (this != &other) {
      Picture tmp(other);
      using std::swap;
      swap(*this, tmp);
    }
    return *this;
  }

  void draw(Canvas& canvas) const;
};
```

**After, using `polymorphic`**

```
class Canvas;

class Shape {
 protected:
  ~Shape() = default;

 public:
  virtual void draw(Canvas&) const = 0;
};

class Picture {
  std::vector<polymorphic<Shape>> shapes_;

 public:
  Picture(const std::vector<polymorphic<Shape>>& shapes)
      : shapes_(shapes) {}

  // Picture(const Picture& other) = default;

  // Picture& operator=(const Picture& other) = default;

  void draw(Canvas& canvas) const;
};
```